Python for Data Science & AI

Dasun Athukoralage

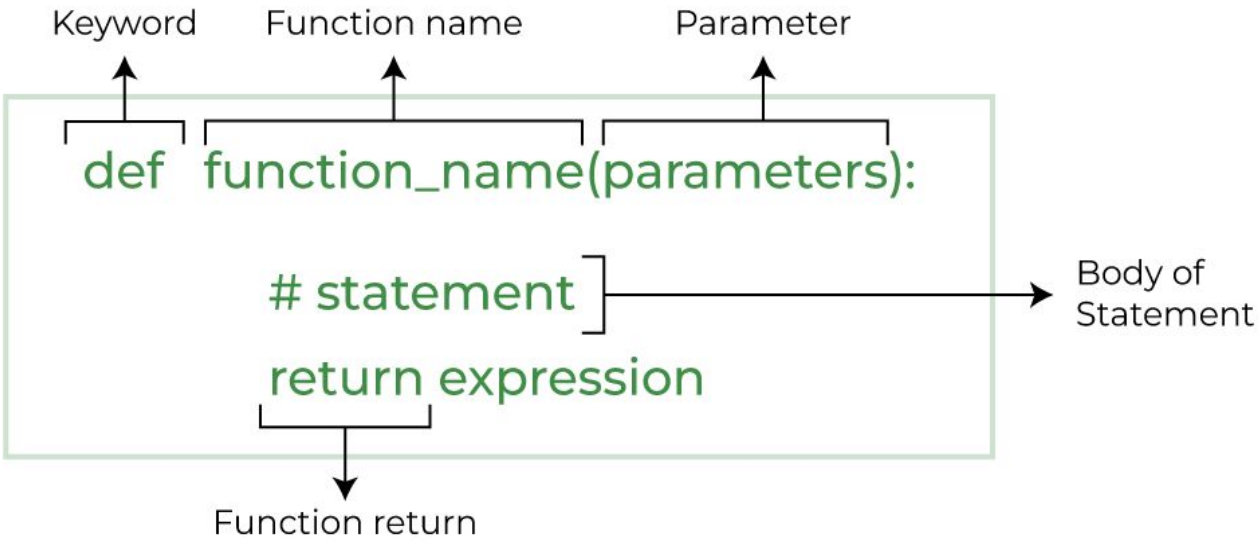
web: www.dasuna.me
email: dasun@nirvanaclouds.com

Functions & Functional Programming

What is a function?

A function in programming is **a reusable block of code that performs a specific task**. It's like a mini-program within a larger program that you can call to execute a certain action or calculation. Functions help in organizing code, making it more modular and easier to understand.

Python function declaration



Python function declaration

– – –

def - keyword used to declare a function

function_name - any name given to the function

parameters - any value passed to function

return (optional) - returns value(s) from a function

Here is an example

```
def add_numbers(a, b):
  """This function adds two numbers."""
  return a + b

print(add_numbers(1, 2))
```

Calling a function in python

In the previous example, we have declared a function named add_numbers(a, b).

Now, to use this function, we need to call it.

Here's how we can call the add_numbers(a, b) function in Python.

```
add_numbers(1, 2) #call the function
```

Python function arguments

Arguments are the values passed inside the parentheses of the function. A function can have **any number of arguments** separated by a comma.

In the previous example, we called the function as follows.

```
add_numbers(1, 2)
```

Here 1 and 2 are arguments.

Types of Python function arguments

In Python, we have the following 4 types of function arguments.

- **Positional arguments**
- **Default arguments**
- **Keyword arguments (named arguments)**
- **Arbitrary arguments (variable-length arguments *args and **kwargs)**

Positional arguments

Positional arguments are a type of function arguments in Python where **the values are passed to the function based on their position or order** in the function's parameter list.

These are the most basic and common type of arguments used when calling a function. When you pass values to a function without explicitly specifying parameter names, Python assigns them to the parameters in the order they appear in the function's parameter list.

Positional arguments

– – –

```
def greet(name, greeting):

    print(greeting, name)



greet("Alice", "Hello")  # Output: Hello, Alice!
```

Default arguments

A default argument is a parameter that **assumes a default value** if a value is not provided in the function call for that argument.

```
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"


print(greet("Alice"))              # Output: Hello, Alice!
print(greet("Bob", "Hi"))          # Output: Hi, Bob!
```

Default arguments

Any number of arguments in a function can have a default value. But once we have a default argument, **all the arguments to its right must also have default values**.

✅ **Valid Example**

```
def func(a, b=10, c=20):
    print(a, b, c)
```

❌ **Invalid Example**

```
def func(a=5, b, c=10):
    print(a, b, c)
```

Keyword arguments

The idea is to allow the caller to specify the argument name with values so that the caller **does not need to remember the order of parameters**.

```
# Python program to demonstrate Keyword Arguments
def student(first_name, last_name):
    print(first_name, last_name)

# Keyword arguments
student(first_name='John', last_name='Nash') # Output: John Nash
student(last_name='Nash', first_name='John') # Output: John Nash
```

Arbitrary Arguments

– – –

Mainly there are two types.

1.) **Arbitrary positional arguments** (*args)

2.) **Arbitrary keyword arguments** (**kwargs)

Arbitrary Positional Arguments (*args)

In Python, arbitrary positional arguments are a special type of parameter that allows a function to accept any number of positional arguments. The asterisk (*) is used to denote an arbitrary positional argument.

```
def arbitrary_positional_args(*args):
  """This function takes an arbitrary number of positional arguments."""
  for arg in args:
    print(arg)
arbitrary_positional_args(1, 2, 3, 4, 5)
```

Arbitrary Positional Arguments (*args)

– – –

Arbitrary positional arguments can be useful when you do not know in advance how many positional arguments will be passed to a function.

For example, you could use arbitrary positional arguments to write a function that sums up a list of numbers, regardless of how many numbers are in the list.

Arbitrary Positional Arguments (*args)

— — —

```
def sum_all_numbers(*numbers):
  """This function sums up an arbitrary number of numbers."""
  total = 0
  for number in numbers:
    total += number
  return total
print(sum_all_numbers(1, 2, 3, 4, 5))
```

Arbitrary Keyword Arguments (**kwargs)

In Python, arbitrary keyword arguments are a special type of parameter that **allows a function to accept any number of keyword arguments**. The double asterisk (**) is used to denote an arbitrary keyword argument.

Arbitrary Keyword Arguments (**kwargs)

— — —

```
def arbitrary_keyword_args(**kwargs):

  """This function takes an arbitrary number of keyword arguments."""

  for key, value in kwargs.items():

    print(f"{key} = {value}")

arbitrary_keyword_args(name="John", age=30, city="New York")
```

Arbitrary Keyword Arguments (**kwargs)

– – –

Arbitrary keyword arguments can be useful **when you do not know in advance how many keyword arguments will be passed** to a function.

For example, you could use arbitrary keyword arguments to write a function that formats a string, regardless of the keyword arguments that are passed to it.

Arbitrary Keyword Arguments (**kwargs)

— — —

```
def format_string(**kwargs):
 """This function formats a string using arbitrary keyword arguments."""
 string = ""

 for key, value in kwargs.items():
   string += f"{key} = {value}, "
 return string

print(format_string(name="John", age=30, city="New York"))
```

Python Built in Functions

What are Built in functions?

Built-in functions are functions that are **pre-defined** in the Python programming language. They are available to use without having to import any external modules.

Why use Built in functions?

Built-in functions are already provided by Python's core libraries, which means you can use them directly without having to write the code for those functions yourself.

This saves time and effort, **promotes code reusability, and follows the DRY (Don't Repeat Yourself) principle**.

Advantages of Built in functions

- **Efficiency**: Built-in functions are typically optimized for performance and efficiency.
- **Consistency**: Since they are part of the standard library, they follow consistent naming conventions and behavior.
- **Well-Tested**: Built-in functions are thoroughly tested and maintained by the Python community.
- **Widely Known**: Many programmers are familiar with these functions, making code more readable and understandable.

Built in functions

The complete list of Built in functions is available here:

https://docs.python.org/3/library/functions.html

Let’s discuss some of the important functions.

Built in functions

---

abs():

Returns the absolute value of a number.

```
abs(-20) # output is 20
abs(20) #  output is 20
```

Built in functions

---

map(): Use to **apply a given function to every item in an iterable** (such as a list, tuple, or other sequence) and **return an iterator that contains the results**.

It's a convenient way to perform an operation on each element of a collection without the need for explicit loops.

Built in functions

---

map():

Syntax:

```
map(function, iterable_1, iterable_2,....)
```

- **‘function’:** The function that you want to apply to each item in the iterable.
- **‘iterable_1’:** The sequence of elements to be filtered.

Built in functions

---

map():

Example 1:

Suppose you have a list of numbers and you want to square each number using the **map()** function.

Built in functions

---

map():

Example 1:

```
numbers = [1, 2, 3, 4, 5]

def square(x):

    return x ** 2

squared_numbers = map(square, numbers)  #<class 'map'>

print(list(squared_numbers))  # Output: [1, 4, 9, 16, 25]
```

Built in functions

---

map():

Example 2: Using **map()** with Multiple Iterables

```
numbers_1 = [1, 2, 3]

numbers_2 = [10, 20, 30]

def add(x, y):

    return x + y

result = map(add, numbers_1, numbers_2)

print(list(result))  # Output: [11, 22, 33]
```

Built in functions

---

map():

Example 3: Show how you can use the map() function to convert the following string list to an integer list.

```
strings = [“1”, “2”, “3”, “4”, “5”]
```

??????????????

Built in functions

---

filter():

filter elements from an iterable (such as a **list, tuple, or other sequence**) based on a specified condition. It **creates a new iterable** containing only the elements that satisfy the given condition.

The filter() function **takes two arguments**: a function that defines the filtering condition and the iterable to be filtered.

Built in functions

---

filter():

Syntax:

filter(function, iterable)

- **‘function’:** A function that returns **True** or **False** for each element in the iterable. **Only elements for which the function returns True are included in the output**.
- **‘iterable’:** The sequence of elements to be filtered.

Built in functions

---

filter():

Example 1:

Suppose you have a list of numbers and you want to filter out the even numbers.

Built in functions

---

filter():

Example 1:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def is_even(x):

    return x % 2 == 0

even_numbers = filter(is_even, numbers) #<class 'filter'>

print(list(even_numbers))  # Output: [2, 4, 6, 8, 10]
```

Built in functions

---

filter():

***Example 2*: Filtering ‘None’ and ‘Empty’ values**

If you want to filter out None, empty strings, and similar values, you can use bool as the filtering function.

```
values = [0, 1, "", "hello", None, "world"]

filtered_values = filter(bool, values)

print(list(filtered_values))    # Output: [1, "hello", "world"]
```

Python Lambda Functions

Python lambda functions, also known simply as "**lambda expressions**" or "**anonymous functions**," are small, inline, and unnamed functions that can be used to create simple, **one-line functions** on the fly.

Lambda functions are often used for tasks that require a short, throwaway function **without** the need to define a formal function using the **def keyword**.

Python Lambda Functions

Syntax:

```
lambda arguments: expression
```

- **lambda**: The keyword that indicates the creation of a lambda function.
- **arguments**: The input parameters for the function, similar to regular functions.
- **expression**: The single expression that the lambda function will evaluate and return.

Python Lambda Functions

Example:

```
add = lambda x, y: x + y

result = add(5, 3)  # Result will be 8
```

How to call without assigning to a variable?

```
result = (lambda x, y: x + y)(5, 3)  # Result will be 8
```

Python Lambda Functions

Limitations:

- They cannot contain statements or multiple expressions.

```
f = lambda x: print(x) # SyntaxError: cannot use statement in lambda

f = lambda x: (x + 2; x * 3) # SyntaxError: invalid syntax
```

- They are less readable for complex operations compared to well-named functions.

Why use lambda functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Let's say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Why use lambda functions?

```
# Let's use that function definition to make a function
that always doubles the number you send in.

mydoubler = myfunc(2)

print(mydoubler(13))
```

Why use lambda functions?

```
# Use the same function definition to make a function that
always triples the number you send in.

mytripler = myfunc(3)

print(mytripler(13))
```

Python lambda with if else

```
# Example of lambda function using if-else


Max = lambda a, b : a if(a > b) else b

print(Max(1, 2))
```

Using filter() with lambda expressions

Lambda functions are often used with the **filter()** function to define filtering conditions concisely.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


even_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even_numbers))  # Output: [2, 4, 6, 8, 10]
```

Thank You...!


“LOTS OF COMPANIES DON'T SUCCEED OVER TIME. WHAT DO THEY FUNDAMENTALLY DO WRONG? THEY USUALLY MISS THE FUTURE.”
Larry Page
Co-FOUNDER, Google